Threads and its Types

Difference between Fork() and Threads

Both fork() and threads are used for multitasking, but they work differently.

Feature	Fork (Creates Process)	Thread (Creates Lightweight Process)
Definition	Creates a new child process (a copy of the parent).	Creates a new thread inside the same process.
Memory Sharing	Parent and child do not share memory.	Threads share memory with the main process.
Communication	Needs Inter-Process Communication (IPC).	Can easily share data (since they share memory).
Execution Speed	Slower (as OS needs to allocate separate resources).	Faster (less overhead since threads share memory).
System Call Used	fork() system call.	pthread_create() in C (POSIX threads).
Crash Impact	If the child crashes, the parent keeps running.	If one thread crashes, it may affect the whole process.

```
Parent Process
     |
     ├── Fork() → Creates Child Process 1
     |
     ├── Fork() → Creates Child Process 2
     |
     ├── Fork() → Creates Child Process 3
```

Fork

```
Main Process (Single Memory Space)
      |
      ├── Thread 1 (Task A)
      |
      ├── Thread 2 (Task B)
      |
      ├── Thread 3 (Task C)
```

Thread

Threads and its Types

Imagine your computer is like a busy restaurant kitchen. To get everything done quickly, you need different people doing different tasks at the same time. That's where threads come in!

A **Thread** is like a helper working inside the process. Threads divide the work of a process into smaller tasks. A process can have multiple threads, all running independently but sharing the same memory space.
Imagine a web browser:

- One thread handles UI interactions.
- Another thread loads web pages.
- A third thread downloads files in the background.

Types of Threads in OS

1. User-Level Threads (ULT)
2. Kernel-Level Threads (KLT)

Threads and its Types

1. **User Level Threads**: These threads are managed by the application itself, not the operating system.

- They are faster to create and manage because they don't involve the OS.
- However, if one user thread blocks (waits), all other user threads in the same process might also block.

2. **Kernel Level Threads**: These threads are managed directly by the operating system.

- They are slower to create and manage because they require OS intervention.
- If one kernel thread blocks, other kernel threads in the same process can still run.
- Kernel threads are more powerful, and allow for true parallel processing on multicore systems.

Threads and its Types

The OS doesn’t recognize user threads directly. If one thread blocks, all threads in that process stop. So it can't utilize multiple CPU cores properly.

Multithreading models are needed to:
1. Optimize thread performance
2. Reduce system overhead
3. Utilize multi-core processors

Multithreading Models (How ULT & KLT Work Together)

1. **Many-to-One** : Multiple user threads map to one kernel thread (Not efficient).
2. **One-to-One** : Each user thread maps to one kernel thread (Used in modern OS).
3. **Many-to-Many** : Multiple user threads map to multiple kernel threads (Best of both).

```
User Threads:   [ T1 ]    [ T2 ]    [ T3 ]
                     \      |      /
                      \     |     /
                        [ Kernel Thread ]
```

Many-to-One

```
User Threads:    [ T1 ]    [ T2 ]    [ T3 ]
                      |         |         |
Kernel Threads:  [ K1 ]    [ K2 ]    [ K3 ]
```

One-to-One

```
User Threads:   [ T1 ]   [ T2 ]   [ T3 ]   [ T4 ]
                     \      |      /        |
                      \     |     /         |
Kernel Threads:  [ K1 ]   [ K2 ]   [ K3 ]
```

Many-to-Many